MultiLex,

A Pipelined Lexical Analyzer

Timothy Bickmore
Robert E. Filman

Software Technology Center
Lockheed Martin Missiles & Space O/96-10 B254E
3251 Hanover Street
Palo Alto, California 94304

415-354-5250
415-424-2999 (FAX)
filman@stc.lockheed.com

Abstract

MultiLex is a lexer generator designed to facilitate creation of lexical analyzers, particularly lexical analyzers for LALR(1) parsers of legacy languages. Innovative features of MultiLex include its pipeline architecture, lexical pattern-matching, manipulation of a larger space of objects than just characters, reconfigurability for languages that include sub-languages, and lexically-scoped dictionary mechanism. We discuss the place of lexers in reengineering of legacy languages, the features of MultiLex, and compare it to prior work on lexers.

Keywords: Lexers, Software reengineering, legacy programming languages, MultiLex, scanners, regular expressions, parsing

Introduction

Lockheed InVision has developed a collection of tools for software reengineering: analyzing software systems to aid human understanding, transforming systems to new environments, and recasting systems to new architectures and languages.[1] The foundation of this activity is a collection of *language workbenches*. A workbench has procedures for parsing source code into an internal, object-based representation and manipulating that representation. Key to reengineering is quickly assimilating additional languages into our tool set, particularly legacy languages that lack modern analysis tools.

A critical element of a language workbench is the lexer. The lexer takes the source character stream and produces a stream of tokens for the parser. Our environment uses LALR(1) parsing (e.g., yacc[2]) Few languages have natural LALR(1) grammars, so the task of looking ahead in the input falls to the lexer.

The most popular way to write lexers has been with lexer-generators such as lex.[3] Lex takes lexical patterns stated as *regular expressions* on characters and actions associated with those patterns. It produces a lexer that recognizes the patterns and performs the actions. The actions generate tokens for the parser.

Legacy languages pose additional challenges for parsing and lexing, including the need to preserve comments in the finished parse structure (while not cluttering the grammar with explicit comment clauses), handle include files, modify parse behavior based on the declarations in the program, both expand and preserve (perhaps lexically-scoped) macros, record where in the input a particular token was found, deal with column-sensitive and line-oriented languages, and even change lexical rules in different parts of a program (for example, for embedded assembly language). Similarly, the very non-LALR(1)-ness of legacy languages can often require considerable lexer look-ahead.

This paper describes MultiLex, a lexical–analyzer designed to simplify generating lexers for LALR(1) parsers of legacy languages. Key elements of MultiLex include:

- Lexing is performed over a space of objects, not characters. Objects have *values* and *attributes* that can be characters or more complex types.
- Pattern matching of regular expressions over objects.
- A lexer is a pipeline of *translators*. Each translator reads a series of input objects and produces a series of output objects for the next translator (or the parser). A translator, at any given step, consumes a (possibly empty) sequence of input objects, modifies their values and attributes, and produces a (possibly empty) sequence of output objects.
- A given translator in the lexical pipeline can temporarily (or permanently) reconstruct the pipeline to receive an alternative input or perform alternative lexing. This mechanism deals with macro expansion, include files, and embedded sub-languages (e.g. assembler).
- The lexer includes a *dictionary* mechanism for scoped organization of analysis data.

Although our version of MultiLex is implemented in Lisp, these same concepts can be applied in other environments. We have used MultiLex to implement the lexers of several language workbenches, including Jovial J3, CMS2–Y, and Prime Infobasic. We have found that MultiLex substantially simplifies lexer construction and improves lexer maintainability.

Motivation

Lexers perform several levels of analysis concurrently. They recognize and process strings, comments, and macros and synthesize more complex tokens, such as keywords, numbers, and identifiers. The primary motivation for developing a multi-phase lexer was the desire to untangle these multiple levels of lexical analysis. This allows quicker development and simpler maintenance than the monolithic algorithms of regular-expression or hand-coded lexers.

The lexer produces tokens for consumption by the parser. These processes can communicate by returning artificial *pseudo-terminals*, rather than keywords or semantically native terminals and by sharing global data structures. (In this paper, keywords that include "@" are such pseudo-terminals.) Making the parser's state table and current state globally available allows the lexer to check whether a particular terminal is legal at the current parse point.

In developing legacy language workbenches, we found our parsers placing fairly complex contextual requirements on their lexers. This is due to both the hand-coded, "I know where I am in the parse" style of writing legacy compilers and the difficulties of defining semantically meaningful, straightforward LALR(1) grammars for legacy languages. Examples of such requirements, drawn from some of our workbenches, include:

- The lexer must invert the following pairs:

identifier **SYS-DD**	⇒ **SYS-DD** *identifier*
identifier **SYS-PROC**	⇒ **SYS-PROC** *identifier*
identifier **SYS-PROC-REN**	⇒ **SYS-PROC-REN** *identifier*

  (LALR(1) grammars are sensitive to different statements that start out the same way; it is easier to achieve LALR(1)-ness by knowing which kind of statement one is parsing early. Having a variety of statements that all start with arbitrary identifiers complicates this task.)

- The lexer must recognize labels (identifiers at the beginning of a statement followed by a "**.**"). Labels are returned as a single distinct lexical entity. Several labels in a row form an individual label. (Once again, we avoid starting statements with arbitrary identifiers.)

- When the lexer sees "*label* **end**," it needs to produce "*label* **$ end.**" [The *label* shadows the look-ahead to the **end**. Turning the *label* into its own (albeit empty) statement alleviates the problem.]

- When the lexer sees two or more "**$**"s or two or more "**then**"s in a row it needs to eliminate all but the first one. When the lexer sees "**$ then**" it needs to eliminate the “**then**” and then rescan the “**$**.” (Grammars with empty right-hand-side rules are more difficult to make LALR(1).)

- When keyword “**@character-string”** is legal, the lexer must package all characters to the next right parenthesis into a string-terminal. (Sometimes literals have context-sensitive delimiters.)
- If the keyword “**@odd-numer-of-ints-ahead**” is legal, the lexer is to count the number of integers immediately ahead in the input. If the answer is odd, it returns this keyword. (Look-ahead requirements may be unbounded.)
- If the keyword “**@is-like-table**” is legal, the lexer needs to look ahead to see if the thing just before the next “**$**” is the keyword “**L**.” (Once again, legacy languages are often not LALR(k) for any k.)
- The tokens “<” and “>”are both relational and subscript operators. The lexer is responsible for determining which kind each use is. (Lexer look-ahead may be not only unbounded but also heuristic. The language is typeless and “$x < y < z >$” is ambiguous, legal, and actually used.)
- End-of-line is used for statement termination. However, the grammar admits other end-of-lines, and the compiler accepts still others. (The lexer must perform error correction.)
- The lexer should recognize arrays and function names and insert the keyword “@**paren-id**” after them. (The lexer must be sufficiently aware of the declaration structure to know what declarations are in scope.)
- The lexer must recognize and expand macros. (Running a macro-expansion pass over the code before parsing is inappropriate for language workbenches, as (1) macros themselves may be lexically scoped—the expansion pass would need to do something resembling parsing to determine the scope; and (2) the macro-origin of information is important for reengineering.)
- Lexer must preserve comments in some structure accessible to the parse tree, but comments cannot appear explicitly in the grammar. (Since any whitespace can hold a comment, comments shadow the look-ahead and preclude LALR(1) parsing.)
- The lexer is to ignore characters in the first ten columns of each card. (Lexing can be column-sensitive.)
- The language includes embedded assembly language, with an entirely different set of lexical conventions. The lexer must recognize the start and end of embedded assembler and tokenize the assembler appropriately.

We note that these requirements place certain general demands on our lexers:

1. Recognizing patterns (i.e., regular-expressions) on the characters of the input.
2. Taking arbitrary actions (including generating tokens for the parser, ignoring input, and adjust the lexer’s internal state) on pattern recognition.
3. Recognizing patterns at the level of tokens (for example, recognizing token sequences requiring rearrangement.)
4. Remembering (and appropriately forgetting) things about the identifiers in the language.

5. Changing lexical styles (e.g., lexing assembler) and lexical input (e.g., processing macro expansion by pretending to read the macro).

Lexer-generators like lex provide a mechanism for translating regular expressions over a character stream into (token producing) actions, the first two of these. Flex also includes a mode mechanism that enables the fifth. MultiLex provides regular expression patterns to accomplish the third, and a pipeline of lexers to organize the different kinds of patterns being discriminated. Finally, MultiLex provides several facilities (e.g., dictionaries, column predicates) particularly convenient for the fourth and fifth.

On the other hand, the full generality of regular expressions provides unused functionality—the implicit backtracking of Kleene operators. In our experience (which echoes that of Horspool[4]), programming languages are lexically deterministic with respect to iteration. Our first version of MultiLex had a variety of backtracking facilities that have atrophied from disuse. We are left with only backtracking from explicit disjunction.

MultiLex implementation

A MultiLex *lexer* is a pipeline of *translators*. The streams between translators are buffered. Each translator is a (primarily) a series of pattern-action rules. If the pattern of a rule matches the translator's input, the action of that rule is executed. Patterns can include both *predicate* (function-calling) tests and *regular expressions* that must match the input. Pattern matching implicitly binds names to the parts of the pattern. Actions use these bindings, the attributes of bound objects, and arbitrary computation to produce output objects and set their properties. Lexers and translators also have local variables and dictionaries that store processing information and definitions that describe subpattern elements.

A lexer is defined by five elements: an *initial configuration* of translators, a set of variables local to that lexer but shared by its translators (*global variables*), a set of *dictionaries*, a list of the additional translators used by the lexer (that are not in the initial configuration) and an *initialization expression*.

MultiLex Objects

Translators pass streams of *objects*. A MultiLex object is a set of pairs, matching attribute names to values, coupled with a notion of *historical inheritance*. Typical attributes include:

`value`	The "base value" of the object. This is the value used in the pattern-matching of translator rules.
`type`	The type of `value`. Pattern-matching predicates can refer to the type of a value.
`objects`	A sequence of the objects matched in the pattern that built this object.

`source`	Name of the source file or function, or the actual input string.
`line`	The line number of this object in the source.
`column`	The column number of this object in the source.

Users can define arbitrary additional attributes.

MultiLex value inheritance is historical because a sought attribute, if not found on the current object, is looked for recursively on the first of that object's `objects`. Typically, objects fed into the lexer pipeline have attributes such as `column`, `source`, and `line`. They are retrieved from conceptually higher-level objects through historical inheritance.

Translators

A *translator* reads an object stream and writes an object stream. It can create new objects and pass input objects unchanged, modified or not at all. For example, a translator can tokenize a character stream by translating it into a sequence of objects that describe the words found. Translators have local variables, local definitions and a sequence of local rules. Each definition associates a name with either a pattern or computable predicate, allowing that name to be used in patterns. Each rule is a pair, consisting of a pattern (or pattern and test) and a series of actions. A rule (or subsequence of rules) may be modified with a conditional test.

In a translator, each grammar-rule–name and local variable defines a local variable. When invoked for an output, the translator sequentially examines its rules until it finds one that matches its input. If so, the rule-names used in the pattern are bound (respectively) to objects that include the objects they matched. The actions of the matching rule execute. A rule within the scope of a conditional is only tested if that conditional is true.

Rule actions produce objects. Typically, these objects are named by translator definitions. The default values of these new objects are the sequence of the values of the objects that were consumed in matching that pattern; the objects of these new objects are a list of the source objects. Often a lexer rule coerces the sequences of values to a conceptually higher element (e. g., coercing a sequence of digits to an integer.) Every successful pattern match creates an object bound to `token`, corresponding to the entire match.

We chose this organization because (1) It allows identifying the parts of a pattern match by name. This is more intuitive than the numeric approaches in systems such as Lex and Emacs.[5] (2) It permits mixing pattern-matching with arbitrary predicate evaluation, critical for legacy languages with complex conditions in their lexical analysis. (3) It enables modes and avoids pointless pattern matching through the use of conditional clauses.

Regular Expressions

MultiLex's regular expression language extends a conventional regular expression language with additional operators germane to MultiLex tokenization: column operators, object matching, and type-based matching. In regular expressions, "@*m*:*n*" matches only if the input is between columns *m* and *n*; "<<*typename*>>" matches only an object of the given type, and "<<*typename value*>>" matches only an object of the given type with the specified literal value.

Figure 1 shows a simple example, the translator MyTrans. This translator defines patterns for letters, digits, and words. It has four rules. The first matches any object of type `comment`, passing this object unchanged to the next translator. The second recognizes strings of digits as integers, computes the integer's numeric value, and passes a token with this value. The third recognizes "quoted" words as `symbols`, when symbols are legal and the discovered symbol passes the `good-symbol?` test, and the fourth, unquoted words as `identifiers`. (The `produce` operator takes parameters describing how to coerce its sequence of objects to a new value and what type to assign that new value. The example is in an "ALGOL-like" publication-language, sparing the reader the effort of dealing with our actual Lispish notation.)

```
translator MyTrans
   definitions
      <digit>  = "[0..9]";
      <letter> = "[A..Za..z]";
      <word>   = "<letter>{<letter>|<digit>}*";
   rules
      "<<comment>>"  => produce (token.0);
      "<digit>*"     => produce (token, seq-to-integer, integer);
      when legal?(symbol)
         "'<word>" & good-symbol? (word)
                     => produce (word, seq-to-symbol, symbol);
      end when;
      "<word>"       => produce (word, seq-to-ident, identifier);
end translator;
```

Figure 1: A simple MultiLex translator

Other features

Several other features of MultiLex are worth mentioning. A sequence of one or more translators is a *configuration*. A translator can change the overall configuration by replacing itself with another configuration. This can be done either in a state-preserving fashion (like a subroutine call) or unconditionally (like a goto). This mechanism is useful for handling multiple languages (e.g., a high-level language with embedded assembler) in the same input and for feeding macro expansions and include files back into the input stream.

MultiLex operates on a stream of objects. The system provides functions for transforming an input file or string into such a stream and for specifying the default attributes of the objects so created (e.g., `line`).

MultiLex includes scoped *dictionaries* that store key/value pairs. Typical uses of dictionaries are for recording macro bodies and for remembering which identifiers have particular attributes, (e.g., being types or matrices.)

MultiLex is implemented as a Common Lisp program. The system includes a number of debugging and parser interface functions not described in this paper, and a compiler that compiles lexers and translators to Lisp. Similar ideas could be applied to compile to conventional imperative languages.

Limitations of this approach

Limitations of MultiLex include:

- We haven't solved the "extra-syntactic" problem. Legacy system workbenches must retain in the parse structures extra-syntactic information such as comments, source lines and files, and macro origins of code. MultiLex has no magical answer to the integration of this information into parse tree. Including things such as comments in parse trees produces trees with the wrong kinds of information in the wrong places. Our approach has been to annotate certain terminals with the extra information, for example, associating comments with identifiers or statement terminators.
- MultiLex is slower than single-stage lexers. Running through several stages, pattern matching with respect to an unbounded space and dynamically binding names all require computation.
- Pipelines function asynchronously. In particular, inquiring about the parser's state ("Is this keyword legal here?") is usually meaningful only for the first token of the last translator in the pipeline.
- In a similar vein, pipelines force early decisions. Occasionally a downstream translator must decompose elements back to their primitive constituents and re-analyze.

Alternative Approaches

Lex[3] exemplifies the dominant practice for lexical generators. Lex provides regular expressions, including sets, optional elements and Kleene operators, the slash look-ahead operator, definitions, and rules (with greedy and priority matching). Lex is typically integrated with C code that can perform alternative actions and maintain the lexer state. Flex[6] extends lex with modal variables for its rules.

The INDIA Lexic Generator[7] uses a grammar (rather than regular expressions) to define patterns. INDIA builds an FSA and uses it to generate code to perform the pattern matching. The resulting program has no procedure calls (except for symbol table interfaces), and a few other optimizations to enhance run-time efficiency. Similarly, Alex[8] uses an extended BNF form to define the lexical generator.

GRAMOL[9] integrates lexing and parsing using an extended BNF for both. GRAMOL takes these descriptions and outputs a parser and scanner. The scanner generator builds an FSA, the system uses a greedy pattern matcher, the user can specify the number of significant characters in a token (i.e., the first n), case conversion is a primitive, keywords are always preferentially matched over more general patterns (e.g., identifiers), and the system employs context-sensitive ambiguity resolution (e.g., look-ahead: <*token1*> **when_followed_by** <*lexical-expression*> and look-behind: <*token1*> **when_preceded_by** <*token2*>.

Salomon and Cormack[10] present a system that extends context-free grammars with metalinguistic enhancements. This allows them to dispense with lexers, describing language syntax completely at the grammatical level. The resulting system is considerably faster than two process parsers, though they do not report if their approach can be extended to the complexities of legacy languages.

Dyadkin[11] presents an idea similar to the pipelines of MultiLex in a Fortran compiler composed of ten pipelined LL(1) parsers. This system incorporates all the work of each level of parsing into an LL(1) parser; our approach allows intermixing of structured parsing and computational steps, and integrates with yacc-style LALR(1) parsers.

In Mkscan,[4] Horspool and Levy dispense with linguistic representations of lexers, providing instead a pattern-by-example graphical user-interface that compiles to a conventional scanner.

Summary

MultiLex is a lexer particularly appropriate for parsing legacy languages. MultiLex has been successfully used, by several programmers (including at least one who is not an author of the system) for several language workbenches. Novel features of MultiLex include its pipelined architecture, its pattern-matching over a larger space of tokens than simple characters, its ability to reconfigure for alternative modes and sub-languages, and its dictionary mechanisms.

Acknowledgments

Our thanks to Paul Morris for comments on the drafts of this paper.

References

1. R. E. Filman, "Applying AI to software reengineering." *Automated Software Engineering*.
2. S. C. Johnson, “Yacc—Yet another compiler compiler,” CSTR32, Bell Laboratories, Murray Hill, New Jersey, 1975.
3. M. E. Lesk, “LEX–A lexical analyzer generator,” CSTR 39, Bell Laboratories, Murray Hill, New Jersey, 1975
4. R. N. Horspool and M. R. Levy, “Mkscan—An interactive scanner generator,” *Software Practice and Experience*, **17**, 369–378, 1987.
5. R. Stallman, *Emacs Manual,* Free Software Foundation, Cambridge MA, 1989.
6. J. Poskanzer and V. Paxson, *Flex manual*, Free Software Foundation, Cambridge MA, 1990.

7. M. Albinus and W. Abmann, “The INDIA lexic generator,” *Workshop on Compiler Compilers and High Speed Compilation*, Springer Verlag, Berlin, 115–127, 1988.

8. H. Mössenböck, “Alex—A simple and efficient scanner generator,” *SIGPLAN Notices*, **21**, 139–157, 1986.

9. C. Genillard and A. Strohmeier, “GRAMOL—a grammar description language for lexical and syntactic parsers,” *SIGPLAN Notices*, **23**, 103–115, 1988.

10. D. J. Salomon and G. V. Cormack, “Scannerless NSLR(1) parsing of programming languages,” SIGPLAN ‘89 Conference on Programming Language Design and Implementation, *SIGPLAN Notices*, **24**, 170–178, 1989.

11.L. J. Dyadkin, “Multibox parsers,” *SIGPLAN Notices*, **29**, 54–60, 1994.